Yannick Renard

MANUEL D'UTILISATION Sage POUR LA CRYPTOGRAPHIE SYMETRIQUE

décembre 2010

Table des matières

Chapitre 1. Présentation du logiciel Sage. 2
1.1. Introduction. 2
1.2. Contexte. 2
1.3. Historique. 3
1.4. Modes d'utilisation. 3
1.5. Langage de programmation. 3
1.6. Composants de Sage. 3
1.7. Utilisation de Sage en cryptographie. 4
1.8. Extension de Sage pour la cryptographie. 4
Chapitre 2. Manipulation des corps finis avec Sage. 5
Chapitre 3. Manipulation de S-Box avec Sage. 8
Chapitre 4. Manipulation d'anneaux polynomiaux multivariés avec Sage. 10
Chapitre 5. Manipulation de LFSR avec Sage. 14
Chapitre 6. Manipulation de fonctions booléennes avec Sage. 16
Bibliographie 20

Chapitre 1

Présentation du logiciel SAGE.

1.1. Introduction.

Le logiciel Software for Algebra and Geometry Experimentation (SAGE) est un logiciel de mathématiques qui rassemble de nombreux programmes et bibliothèques libres dans une interface commune basée sur le langage de programmation Python.

Il a pour ambition de devenir une alternative libre aux logiciels propriétaires comme Magma, Maple, Mathematica ou Matlab.

Il couvre un champ le plus vaste possible, comme l'algèbre, l'analyse, la théorie des groupes, la théorie des graphes, la théories des nombres, l'analyse et la cryptographie.

Les principaux objectifs de SAGE sont d'une part de faciliter l'expérimentation d'objets mathématiques avec un logiciel rapide, convivial, bien documenté et open-source, et d'autre part de favoriser la coopération et la mise en commun des développements réalisés.

1.2. Contexte.

SAGE s'inscrit dans un environnement de logiciels dédiés aux mathématiques déjà nombreux, que l'on peut classer en trois catégories :

1. Logiciels propriétaires payants :
   a) Mathematica (`http://www.wolfram.com`),
   b) Maple (`http://www.maplesoft.com`),
   c) Magma (`http://magma.maths.usyd.edu.au/magma`),
   d) Matlab (`http://www.mathworks.com`).
2. Logiciels propriétaires gratuits :
   a) Kash/Kant (`http://www.math.tu-berlin.de/kant`),
   b) CoCoA (`http://cocoa.dima.unige.it`).
3. Logiciels open-source :
   a) GAP (`http://www.gap-system.org`),
   b) Macaulay2 (`http://www.math.uiuc.edu/Macaulay2`),
   c) Maxima (`http://maxima.sourceforge.net`),
   d) Octave (`http://www.octave.org`),
   e) Pari/GP (`http://pari.math.u-bordeaux.fr`),
   f) Singular (`http://www.singular.uni-kl.de`).

La particularité de SAGE est d'être distribué sous licence libre, selon les termes de la GNU General Public License (GPL) version 2 ou ultérieure. Cette dernière autorise la copie, l'étude, la modification, l'amélioration et la distribution de SAGE et de son code source. SAGE intégre la plupart des logiciels open-source et offre également une interface avec les logiciels propriétaires Magma, Maple et Mathematica.

Une comparaison de SAGE avec, entre autres, Mathematica, Maple et Matlab est disponible dans la thèse de M.V.Nguyen.

1.3. Historique.

Le projet SAGE est récent et a commencé en 2004-2005 sous l'impulsion de W.Stein, professeur de l'université de Washington. Des administrations comme le ministère américain de la défense, des entreprises comme Microsoft, Google ou Sun, ainsi que des donateurs privés ont participé financièrement.

La première version SAGE 1.0 a été présentée lors des SAGE Days de San Diego en février 2006.

Actuellement, le logiciel SAGE connaît une nouvelle version toutes les une à deux semaines, enrichi par de nombreux développeurs.

1.4. Modes d'utilisation.

Le logiciel SAGE a plusieurs modes d'utilisation :

1. Il peut être utilisé en ligne de commande.
2. Il peut exécuter des programmes interprêtés ou compilés avec un compilateur Cython, via un fichier *.py ou via un fichier *.spyx .
3. Le mode le plus convivial est l'interface graphique notebook. Ce mode notebook fonctionne avec un mécanisme client-serveur et s'utilise avec un navigateur web. Ce dernier permet de partager les ressources d'une machine ou d'échanger des feuilles de calcul worksheet. Il offre la possibilité d'évaluer directement des expressions comme de programmer des fonctions.

1.5. Langage de programmation.

Au lieu de faire appel à un langage spécifique, le logiciel SAGE utilise un langage de programmation quasiment identique au Python. Les seules différences concernent les symboles de l'exponentiation (^ ou ** en SAGE, ** en Python) et de la division (2/3 est une fraction rationnelle en SAGE, 2/3 est une division entière en Python).

Le Python est un langage de programmation interprêté open source et très répandu. Il offre de nombreux avantages concernant l'enregistrement d'objets, la gestion de la mémoire, la disponibilité de nombreux packages, la portabilité, le débogage et le profilage.

Certaines parties du code et des bibliothèques devant fonctionner rapidement sont écrites en langage compilé.

1.6. Composants de SAGE.

SAGE rassemble une grande quantité de logiciels et de bibliothèques spécialisées sous une même interface.

Il intégre notamment le logiciel d'algèbre GAP, le langage de calcul Octave, le logiciel de statistique R, le logiciel d'arithmétique pari/GP, le logiciel pour l'algèbre commutative et non commutative et de manipulation de polynômes multivariés Singular, le logiciel de calcul formel Maxima.

Il comprend aussi le logiciel GMP-ECM pour la factorisation des entiers en utilisant des courbes elliptiques, une librairie cryptographique généraliste Libgcrypt, ECLib pour l'énumération des courbes elliptiques, le logiciel de cryptographie OpenCDK ou une librairie Python de cryptographie Python Cryptography Tollkit (PyCrypto) implémentant notamment des fonctions de hachage et les algorithmes cryptographiques à clé publique. Par ailleurs, des fonctions de hachage comme MD5 et la famille SHA sont fournies avec la librairie standard de Python.

Le logiciel SAGE offre également une interface avec les logiciels propriétaires Magma, Maple et Mathematica.

La liste complète des composants de SAGE est disponible à l'adresse suivante : `http://sagemath.org/links-components.html`.

1.7. Utilisation de SAGE en cryptographie.

SAGE a été utilisé, dès sa création, comme outil d'expérimentation au profit de la recherche en cryptographie.

Ainsi, M.Albrecht a implémenté sur SAGE son attaque algébrique contre le Toy Cipher de N.Courtois. S.Maitra et S.Sarkar ont utilisé le module de théorie des nombres de SAGE pour compléter l'ensemble des exposants RSA faibles. Un autre exemple est le travail réalisé avec SAGE par G.Bertoni et al. pour proposer le candidat Keccak comme famille de fonctions de hachage SHA-3.

D'autres travaux cryptographiques pour lesquels SAGE a été utilisé ont été réalisés par M.Albrecht et C.Cid, Y.Aner, G.V.Bard, D.J.Bernstein et al., D.Boneh et al., V.Velichkov et al., et R.Weinmann.

1.8. Extension de SAGE pour la cryptographie.

L'un des objectifs de SAGE est de favoriser la coopération et la mise en commun des développement réalisés. Les fonctionnalités de théorie des nombres, corps finis et courbes elliptiques ont été implémentées par l'équipe de développement de SAGE (Sage Development Team) en s'appuyant sur les composants existants de SAGE. Plusieurs modules dédiés à la cryptographie et réalisés par d'autres développeurs ont ensuite complété les classes disponibles dans SAGE.

Ainsi, la plupart des fonctionnalités orientées cryptographie jusque dans la version 3.4 incluse du 11 mai 2009 ont été implémentées par D.Kohel.

Les classes `sage.crypto.mq.sbox` et `sage.crypto.mq.sr` permettant de manipuler des S-box et de paramétrer l'AES ont été réalisées par M.Albrecht.

La classe `sage.crypto.block_cipher.miniaes` pour une variante simplifiée de l'AES appelée mini-AES a été développée par M.V.Nguyen.

La classe `sage.crypto.boolean_function.BooleanFunction`, permettant d'étudier l'immunité algébrique et le spectre de Walsh de fonctions booléennes, a été réalisée par Y.Laigle-Chapuy.

Tous ces apports au logiciel SAGE sont très récents, ils datent de 2009 et 2010. La liste complète des classes SAGE et leur description est disponible dans le logiciel SAGE en mode notebook par le chemin `help/reference manual/global module index`.

Chapitre 2

Manipulation des corps finis avec SAGE.

Les corps finis se définissent dans SAGE à l'aide du constructeur `FiniteField` ou `GF`.

On peut construire les corps premiers $GF(p)$ avec p premier :

```
sage : GF(11)
Finite Field of size 11
sage : [x for x in GF(11)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

On peut aussi construire les corps composés $GF(q)$ avec $q = p^k$, p premier et $k > 1$ un entier. Un corps non premier $GF(p^k)$ avec p premier et $k > 1$ est isomorphe à l'anneau quotient des polynômes de $GF(p)[x]$ modulo un polynôme unitaire et irréductible de degré k. Dans ce cas, il faut préciser, outre le nombre d'éléments, le nom du générateur du corps :

```
sage : K.<a> = GF(2^8,name='a') ; K, K.gen()
(Finite Field in a of size 2^8, a)
sage : K.order(), K.characteristic(), K.degree()
(256, 2, 8)
sage : K.modulus()
x^8 + x^4 + x^3 + x^2 + 1
```

Notons que l'écriture suivante est valide mais ne permet pas de créer des éléments du corps comme polynôme en le générateur a :

```
sage : K = GF(2^8,name='a')
sage : y = a^8 + a^4 + a^3 + a + 1
NameError : name 'a' is not defined
```

Le polynôme irréductible permettant de décrire le corps fini est choisi par défaut par SAGE. On peut aussi le préciser lors de la création du corps, en ayant au préalable défini l'anneau polynomial $GF(p)[x]$:

```
sage : R.<x> = GF(2)['x']
sage : K.<a> = GF(2^8,name='a',modulus=x^8+x^4+x^3+x+1)
sage : K.modulus()
x^8 + x^4 + x^3 + x + 1
```

On peut maintenant considérer des élements du corps $GF(p^k)$:

```
sage : y = a^8 + a^4 + a^3 + a + 1 ; y
0
sage : z = K.random_element() ; z
a^7 + a^6 + a^3 + a
sage : p = x^5 + x^2 +1 ; K(p)
a^5 + a^2 + 1
```

Attention, le modulus est défini à la création du corps fini et l'isomorphisme avec un anneau quotient $GF(p)[x]$ modulo un autre polynôme irréductible de

degré k, n'est pas implémenté sous Sage :

```
sage : L.<b> = GF(2^8,name='b',modulus=x^8+x^4+x^3+x^2+1)
sage : L(a^7+a^6+a^3+a+1)
TypeError : unable to coerce from a finite field other than the
prime subfield
#impossible de passer du corps K au corps L
```

Néanmoins, on peut considérer avec Sage l'isomorphisme canonique entre $GF(p^k)$ et $GF(p)^k$. Lorsque $p = 2$, on pourra choisir pour $GF(p)^k$ une représentation entière ou binaire.

Pour transformer un entier ou une liste de bits de $GF(2)^k$en élément d'un corps fini $GF(2^k)$, on convertit un entier en liste de bits puis en un élément d'un corps fini :

```
sage : R.<x> = GF(2)['x'] ; K.<a> = GF(2^8,name='a')
sage : K(37.digits(2)), K([1,0,1,0,0,1])
(a^5 + a^2 + 1, a^5 + a^2 + 1)
sage : K(37)
1
```

Attention, Sage utilise ici la représentation little endian. Par ailleurs, `K(37)` renvoie 37 modulo la caractéristique du corps K et pas $a^5 + a^2 + 1$!

Une conversion intermédiaire en polynôme est possible mais pas nécessaire pour les corps $GF(2^k)$ avec $k < 16$ (librairie givaro) :

```
sage : K(R(37.digits(2))), K(R([1,0,1,0,0,1]))
(a^5 + a^2 + 1, a^5 + a^2 + 1)
```

Cette conversion intermédiaire en polynôme est obligatoire pour les corps $GF(2^k)$ avec $k \geq 16$ (implémentation ntl) et $GF(p^k)$ avec $p \neq 2$ (implémentation pari) :

```
sage : K = GF(2^16,name='a')
sage : K(37.digits(2))
TypeError : unable to coerce
sage : R = K['x']
sage : K(R(37.digits(2)))
a^5 + a^2 + 1
```

Réciproquement, pour transformer un élément d'un corps fini $GF(2^k)$ en un entier ou une liste de bits de $GF(2)^k$ on convertit un élément d'un corps fini en liste de bits de $GF(2)^k$ grâce à la méthode `vector_space()` puis éventuellement en entier :

```
sage : V = K.vector_space()
sage : list(V(a^5+a^2+1)), ZZ(list(V(a^5+a^2+1)),2)
([1, 0, 1, 0, 0, 1, 0, 0], 37)
```

Il est à noter que la librairie givaro utilise la représentation logarithmique et des tables de Zech. Un corps non premier $GF(p^k)$ avec p premier (pour givaro $p = 2$) et $k > 1$ (pour givaro $k < 16$) peut être représenté par l'anneau quotient des polynômes de $GF(p)[x]$ modulo un polynôme primitif de degré k. Il peut aussi être représenté par $GF(a)$, où tout élément non nul de $GF(a)$ est une puissance de a. L'élément a, appelé générateur, est une racine du polynôme primitif. Avec cette représentation logarithmique, la multiplication s'écrit $a^i * a^j = a^{i+j}$ et l'addition s'écrit $a^i + a^j = a^i(1 + a^{(j-i)}) = a^i * a^{Z(j-i)}$, où les valeurs de $Z(i)$

ont été mises en table appelée table de Zech :

```
sage : y = K._cache._element_log_repr(a^5+a^2+1) ; y
'36'
sage : y = a^36 ; y
a^5 + a^2 + 1
```

On mentionne également l'existence des méthodes suivantes de la librairie givaro (mais on préfèrera celles précédemment évoquées dans un cadre plus général et qui sont valables pour les trois implémentations givaro, ntl et pari) :

```
sage : K.<a> = GF(2^8,name='a')
sage : y = Integer(K._cache._element_int_repr(a^5+a^2+1)) ; y
37
sage : y = K._cache._element_poly_repr(a^5+a^2+1) ; y
'a^5 + a^2 + 1'
```

Pour être complet, la documentation et le code source de la classe `Finite FieldFactory` sont obtenues respectivement par les commandes `GF?` et `GF??` qui ne détaillent pas les méthodes implémentées. Le manuel de référence permet d'accéder à la liste des fonctions et en fournit une description détaillée mais il vaut mieux avoir idée du nom recherché !

En fait, SAGE utilise trois implémentations différentes des corps finis selon leur caractéristique et leur degré :

1. caractéristique 2 et degré < 16 : implémentation givaro
2. caractéristique 2 et degré ≥ 16 : implémentation ntl
3. caractéristique $\neq 2$: implémentation pari

Les documentations et sources respectives sont accessibles par les commandes suivantes :

1. `sage.rings.finite_rings.element_givaro??`
2. `sage.rings.finite_rings.element_ntl_gf2e??`
3. `sage.rings.finite_rings.finite_field_ext_pari??`

La liste des méthodes disponibles peut aussi être obtenue rapidement en créant un corps fini K, puis en utilisant la complétion de ligne de commande. Il suffit alors de taper `K.` et d'appuyer sur la touche tabulation.

Chapitre 3

Manipulation de S-Box avec SAGE.

Une boîte de substitution ou S-Box est un élément de base de la cryptographie symétrique. Elle prend m bits en entrée et renvoie n bits en sortie. La représentation entière est souvent utilisée, de sorte qu'une S-Box est décrite par sa table, qui est une liste de ses images , c'est à dire 2^m entiers appartenant à l'ensemble $[0, 2^n - 1]$.

Dans SAGE, le module `sage/crypto/mq/sbox.py` permet de construire facilement des S-Box à partir d'un tuple d'entiers.

```
sage : S = mq.SBox(0,1,3,6,5,2,4,7)
```

Par défaut, la représentation de la S-Box, vue comme une fonction booléenne vectorielle, est big endian (le bit de poids faible est à droite).

```
sage : print S(3),S([0,1,1]).
6 [1, 1, 0]
```

La commande suivante est équivalente :

```
sage : S=mq.SBox(0,1,3,6,5,2,4,7,big_endian = True)
sage : print S(3),S([0,1,1])
6 [1, 1, 0]
```

On peut aussi préciser la représentation little endian (le bit de poids faible est à gauche) :

```
sage : S=mq.SBox(0,1,3,6,5,2,4,7,big_endian = False)
sage : print S(3),S([1,1,0])
6 [0, 1, 1]
```

On peut considérer la S-Box de l'AES, définie par la fonction inverse (à transformation inversible près) sur le corps $GF(256)$ vu comme l'anneau quotient $GF(2)[x]/x^8 + x^4 + x^3 + x + 1$. Une généralisation paramétrable de l'algorithme AES est implémenté dans SAGE et on peut obtenir la la S-Box AES de la manière suivante :

```
sage : sr = mq.SR(10,4,4,8)
# AES à 10 tours avec une matrice 4*4 sur le corps GF(2^8)
sage : S = sr.sbox() #S est la Sbox AES de GF(2^8) vers GF(2^8)
sage : print S
(99,124,119,123,242,107,111,197,48,1,103,43,254,215,171,118,202,
130,201,125,250,89,71,240,173,212,162,175,156,164,114,192,183,
253,147,38,54,63,247,204,52,165,229,241,113,216,49,21,4,199,35,
195,24,150,5,154,7,18,128,226,235,39,178,117,9,131,44,26,27,110,
90,160,82,59,214,179,41,227,47,132,83,209,0,237,32,252,177,91,
106,203,190,57,74,76,88,207,208,239,170,251,67,77,51,133,69,249,
2,127,80,60,159,168,81,163,64,143,146,157,56,245,188,182,218,33,
16,255,243,210,205,12,19,236,95,151,68,23,196,167,126,61,100,93,
25,115,96,129,79,220,34,42,144,136,70,238,184,20,222,94,11,219,
224,50,58,10,73,6,36,92,194,211,172,98,145,149,228,121,231,200,
```

```
55,109,141,213,78,169,108,86,244,234,101,122,174,8,186,120,37,
46,28,166,180,198,232,221,116,31,75,189,139,138,112,62,181,102,
72,3,246,14,97,53,87,185,134,193,29,158,225,248,152,17,105,217,
142,148,155,30,135,233,206,85,40,223,140,161,137,13,191,230,66,
104,65,153,45,15,176,84,187,22)
```

On peut aussi générer des S-Box aléatoires. Pour cela, on définit un uplet d'entiers :

```
sage : SBox_table = tuple([randint(0,15) for i in range(16)])
sage : print SBox_table
(7, 2, 14, 11, 8, 12, 14, 1, 6, 8, 1, 1, 13, 15, 9, 11)
```

On définit alors la S-Box à partir de sa table :

```
sage : SBox = mq.SBox(SBox_table)
```

On affiche les images de la S-Box :

```
sage : print [SBox(i) for i in range(16)]
[7, 2, 14, 11, 8, 12, 14, 1, 6, 8, 1, 1, 13, 15, 9, 11]
```

On peut aussi générer des permutations aléatoires. Pour cela, on définit une permutation :

```
sage : permutation=Permutations(16).random_element()
sage : print permutation
[8, 14, 7, 2, 13, 1, 4, 12, 6, 11, 16, 15, 5, 3, 10, 9]
```

On définit alors la S-Box :

```
sage : SBox=mq.SBox(tuple([i-1 for i in permutation]))
```

On affiche les images de la S-Box :

```
sage : print SBox
(7, 13, 6, 1, 12, 0, 3, 11, 5, 10, 15, 14, 4, 2, 9, 8)
```

Chapitre 4

Manipulation d'anneaux polynomiaux multivariés avec SAGE.

Les anneaux polynomiaux multivariés se définissent dans SAGE à l'aide des constructeurs `PolynomialRing` ou `InfinitePolynomialRing` ou `Boolean PolynomialRing`. Pour les corps finis, SAGE propose une seule interface et utilise plusieurs implémentations de manière transparente pour l'utilisateur en fonction de la caractéristique et du degré. Ce n'est pas le cas pour les anneaux polynomiaux multivariés. Par exemple, un anneau polynomial multivarié sur $GF(2)$ défini avec `PolynomialRing` utilise l'implémentation LIBSINGULAR alors que `BooleanPolynomialRing` utilise la librairie $C++$ POLYBORI.

La deuxième classe, `infinite_polynomial_ring`, est moins efficace que la première, `polynomial_ring`, mais est plus souple d'utilisation. En effet, elle permet de définir un anneau polynomial avec un nombre d'indéterminées qui augmente suivant les besoins ou avec plusieurs familles de variables. La dernière, `pbori`, est une implémentation optimisée pour les anneaux de polynômes booléens. Le constructeur `PolynomialRing` permet de définir $K[x_1, x_2, ..., x_n]$ où K est un corps, alors que le constructeur `BooleanPolynomialRing` permet de définir l'anneau quotient $GF(2)[x_1, ..., x_n]/ < x_1^2 + x_1, ..., x_n^2 + x_n >$. Cela impliquera notamment que, pour résoudre sur $GF(2)$ un système de polynômes multivariés définis avec `BooleanPolynomialRing`, il n'est pas nécessaire de rajouter les polynômes $x_1^2 + x_1, ..., x_n^2 + x_n$.

Un anneau polynomial multivarié peut être associé à une relation d'ordre total sur ses monômes, appelée ordre monomial. La définition de l'ordre monomial dépend de l'ordre des variables. On considère ici, comme dans SAGE, que $x_0 > x_1 > ... > x_d$. Les principaux ordres monomiaux sont :

1. ordre lexicographique (lex) :
   $x_0^{l_0}...x_d^{l_d} < x_0^{l'_0}...x_d^{l'_d} \iff l_i < l'_i$ pour le plus petit i tel que $l_i \neq l'_i$.
2. ordre lexicographique gradué (deglex) :
   $x_0^{l_0}...x_d^{l_d} < x_0^{l'_0}...x_d^{l'_d} \iff \sum l_i < \sum l'_i$ ou si $\sum l_i = \sum l'_i$, $l_i < l'_i$ pour le plus petit i tel que $l_i \neq l'_i$.
3. ordre lexicographique inversé gradué (degrevlex) :
   $x_0^{l_0}...x_d^{l_d} < x_0^{l'_0}...x_d^{l'_d} \iff \sum l_i < \sum l'_i$ ou si $\sum l_i = \sum l'_i$, $l_i > l'_i$ pour le plus grand i tel que $l_i \neq l'_i$.

Il existe plusieurs méthodes pour définir un anneau polynomial multivarié :

1. On peut définir des variables en les passant en paramètre au constructeur `PolynomialRing` ou `BooleanPolynomialRing` :

   ```
   sage : R.<x,y,z,k> = PolynomialRing(GF(2),'x,y,z,k')
   ou
   sage : R.<x,y,z,k> = BooleanPolynomialRing('x,y,z,k')
   ```
2. On peut définir des variables indicées automatiquement, par exemple 10 variables x_0 à x_9 auquelles on accède par la notation $x[0]$ à $x[9]$. Les variables sont alors ordonnées par ordre décroissant : $x_0 > x_1 > ... > x_9$. L'ordre monomial est optionnel. Attention, la classe `PolynomialRing` utilise l'ordre monomial lexicographique inversé gradué par défaut alors que la classe `BooleanPolynomialRing` utilise l'ordre monomial lexicographique par

défaut.

```
sage : R = PolynomialRing(GF(2),10,'x',order='degrevlex')
sage : x = R.gens()
sage : p = x[1]^2 + x[0]*x[1] ; print p.lt() #leading term
x0*x1
sage : p = x[1]^3*x[4] + x[3]^2*x[4]^2 + x[2]^4 ; print p
x2^4 + x1^3*x4 + x3^2*x4^2
sage : R.term_order()
Degree reverse lexicographic term order

sage : R = BooleanPolynomialRing(10,'x',order='degrevlex')
sage : x = R.gens()
sage : p = x[1]^2 + x[0]*x[1]
sage : print p
x1*x0 + x1
sage : print p.lt()
x1*x0
```

Il faut noter que la ligne `x = R.gens()` est obligatoire pour pouvoir manipuler les variables :

```
sage : R = PolynomialRing(GF(2),10,'x')
sage : print p = x[0] + x[1]
TypeError : x must have length self.ngens()
```

La définition suivante n'est pas valide non plus :

```
sage : R.<x> = PolynomialRing(GF(2),10,'x')
IndexError : the number of names must equal the number of
generators
```

Si on veut manipuler les variables par la notation `xi` plus concise que `x[i]`, il faut utiliser la méthode `inject_variables()` :

```
sage : p = x2 ; print p
NameError : global name 'x2' is not defined
sage : R.inject_variables()
Defining x0, x1, x2, x3, x4, x5, x6, x7, x8, x9
p = x2 ; print p
x2
```

3. Une troisième méthode consiste en la définition d'un anneau polynomial multivarié avec un nombre d'indéterminées qui augmente suivant les besoins :

```
sage : R.<x,y,z,k> = InfinitePolynomialRing(GF(2),
implementation='sparse')
sage : p = x[0] + x[1] + x[2] ; print p
x_2 + x_1 + x_0
sage : p = y[0] + y[1] + y[2] ;print p
y_2 + y_1 + y_0
```

Néanmoins, ce type d'anneau polynomial n'est pas pris en compte par l'implémentation de la mise en équation algébrique de S-Box :

```
sage : S = mq.SBox(7,6,0,4,2,5,1,3)
sage : S.polynomials([y[0],y[1],y[2]],[x[0],x[1],x[2]])
NotImplementedError : Echelon form not implemented over
```

```
'Infinite polynomial ring in x,y,z,k over Finite Field of size 2'.
```

4. Une quatrième méthode repose sur la création d'une liste de variables, obtenue en manipulant les chaînes de caractères, puis passée en paramètre au constructeur `PolynomialRing` ou `BooleanPolynomialRing` :

```
sage : def variable(name,round,bit) :
.... :     variable_list = []
.... :     for i in range(round) :
.... :         for j in range(bit) :
.... :             variable_list.append('%s%03d%03d'%(name,i,j))
.... :     return variable_list
sage : variable_list = variable('k',2,3) ; print variable_list
['k000000','k000001','k000002','k001000','k001001','k001002']

sage : R = PolynomialRing(GF(2),variable_list,order='degrevlex')
ou
sage : R = BooleanPolynomialRing(len(variable_list),
variable_list,order='degrevlex')

sage : R.inject_variables()
```

   Il faut noter que l'ordre de la liste est important puisqu'il définit l'ordre des variables, qui peut influer par la suite sur la résolution de systèmes polynomiaux multivariés. Par ailleurs, cette méthode est avantageuse en terme de lisibilité des variables mais pas en terme de temps d'exécution.

Une fois l'anneau polynomial créé, on peut instancier des polynômes et affecter des valeurs connues à certaines variables :

```
sage : equations_list = [k000000+k000001,k000000+k001002]
sage : equations_filled_list = [p.subs(k000000=0) for p in
equations_list]
sage : print equations_filled_list
[k000001, k001002]
```

Pour affecter plusieurs variables, il faut passer un dictionnaire {variables :valeurs} en paramètre de la méthode `subs()` :

```
sage : equations_filled_list = [p.subs({k000000 :0,k000001 :0})
for p in equations_list]
```

On peut également construire un idéal à partir d'une liste de polynômes :

```
sage : I = R.ideal(equations_list)
```

Pour être complet, les commandes (`Infinite`, `Boolean`) `PolynomialRing?` et (`Infinite`, `Boolean`) `PolynomialRing??` donnent respectivement la documentation et le code source de la classe mais ne détaillent pas les méthodes implémentées. Le manuel de référence permet d'accéder à la liste des fonctions et en fournit une description détaillée mais il vaut mieux avoir une idée du nom recherché !

Les documentations et sources respectives sont accessibles par les commandes suivantes :

1. `sage.rings.polynomial.multi_polynomial_ring??`
2. `sage.rings.polynomial.infinite_polynomial_ring??`
3. `sage.rings.polynomial.pbori??`

La liste des méthodes disponibles peut aussi être obtenue rapidement en créant un anneau polynomial R, puis en utilisant la complétion de ligne de commande. Il suffit alors de taper `R.` et d'appuyer sur la touche tabulation.

Chapitre 5

Manipulation de LFSR avec SAGE.

Un registre à décalage à rétroaction linéaire ou Linear Feedback Shift Register (LFSR) permet d'engendrer une suite pseudo-aléatoire, vérifiant une relation de récurrence linéaire. Un LFSR de n bits peut être représenté par un polynôme de degré n dont les coefficients définissent la relation de récurrence. Il existe deux représentations possibles :

- le polynôme de rétroaction : $1 + \sum_1^n c_i x^i$
  Dans ce cas, la relation de récurrence est $s_t = \sum_1^n c_i s_{t-i} \, mod 2$ et en particulier $s_n = \sum_1^n c_i s_{n-i} \, mod 2$.
  Par exemple, le polynôme de rétroaction $1 + x^3 + x^4$, de coefficients $(c_1, c_2, c_3, c_4) = (0, 0, 1, 1)$ définit un LFSR de 4 bits de relation de récurrence $s_t = s_{t-3} + s_{t-4}$. On a en particulier $s_4 = s_1 + s_0$.

- le polynôme caractéristique : $\sum_0^{n-1} c_i x^i + x^n$
  Dans ce cas, la relation de récurrence est $s_t = \sum_0^{n-1} c_i s_{t-n+i} \, mod 2$ et en particulier $s_n = \sum_0^{n-1} c_i s_i \, mod 2$.
  Par exemple, le polynôme caractéristique $1 + x + x^4$, de coefficients $(c_0, c_1, c_2, c_3) = (1, 1, 0, 0)$ définit un LFSR de 4 bits de relation de récurrence $s_t = s_{t-4} + s_{t-3}$. On a en particulier $s_4 = s_0 + s_1$.

Le polynôme caractéristique est le polynôme inverse du polynôme de rétroaction.
La suite pseudo-aléatoire produite est $s_0, s_1, s_2, ...$
L'état interne à l'instant t est $s_t, s_{t+1}, ..., s_{t+n-1}$.

Du fait des propriétés statistiques obtenues et de l'optimalité en terme de période et de complexité linéaire (taille du plus petit LFSR équivalent), les polynômes de rétroaction utilisés sont primitifs. On peut facilement obtenir des polynômes primitifs avec SAGE :

```
sage : def all_primitive(n) :
.... :     return [p for p in GF(2)['x'].polynomials(
.... :     of_degree=n) if p.is_primitive()==True]
sage : all_primitive(4)
[x^4 + x + 1, x^4 + x^3 + 1]

sage : def primitive(n) :
.... :     generator=(p for p in GF(2)['x'].polynomials(
.... :     of_degree=n) if p.is_primitive()==True )
.... :     return generator.next()
sage : primitive(100)
x^100+x^25+x^24+x^22+x^20+x^19+x^18+x^17+x^16+x^15+x^14+
x^13+x^12+x^11+x^10+x^9+x^8+x^7+x^6+x^5+x^4+x^3+x^2+x+1
```

Dans SAGE, la fonction `lfsr_sequence(key,fill,nb)` produit les nb bits de sortie $s_0, s_1, , ..., s_{nb-1}$ d'un LFSR de taille n défini par les n coefficients (celui de x^n n'est pas compté) de son polynôme caractéristique (`key`) et par son état_initial $s_0, s_1, , ..., s_{n-1}$ (`fill`) représenté par une liste de n éléments de $GF(2)$. La documentation et les sources sont accessibles respectivement par

les commandes `sage.crypto.lfsr?` et `sage.crypto.lfsr??`

Voici un exemple avec le polynôme caractéristique $1 + x + x^4$:

```
sage : coefficients=[GF(2)(i) for i in list('1100')]
#[c0,c1,c2,c3]
sage : initial_state = [GF(2)(i) for i in list('1011')]
#[s0,s1,s2,s3]
sage : l=len(coefficients) #nombre de bits du LFSR
sage : n=2^l-1 #période du LFSR
sage : print lfsr_sequence(coefficients,initial_state,n)
[1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0]
```

L'algorithme de Berlekamp-Massey donne le polynôme de rétroaction du LFSR de taille minimale engendrant une suite. Le LFSR est unique si et seulement si la longueur de la suite est double de la taille du LFSR. Attention, dans SAGE, la fonction `lfsr_connection_polynomial()` donne le polynôme de rétroaction alors que la fonction `berlekamp_massey()` donne le polynôme caractéristique. Voici un exemple, toujours avec le polynôme caractéristique $1+x+x^4$ et de rétroaction $1 + x^3 + x^4$:

```
sage : sequence = lfsr_sequence(coefficients,initial_state,2*l)
sage : print berlekamp_massey(sequence)
x^4 + x + 1
sage : lfsr_connection_polynomial(sequence)
x^4 + x^3 + 1
```

Pour des raisons de simplifications, on propose une nouvelle implémentation du calcul des bits de sortie d'un LFSR à partir directement de l'expression de son polynôme de rétroaction et de son état initial :

```
sage : def lfsr_output(feedback_polynomial,initial_state,n) :
.... :     new_state = [GF(2)(i) for i in initial_state]
.... :     k = len(initial_state)
.... :     c = feedback_polynomial.reverse().coeffs()[0 :k]
.... :     L = []
.... :     for i in range(n) :
.... :         old_state = copy(new_state)
.... :         new_state = new_state[1 :k]
.... :         new_state.append(sum([c[i]*old_state[i]
.... :         for i in range(k)]))
.... :         L.append(old_state[0])
.... :     return L
```

Voici un exemple avec le polynôme de rétroaction $1 + X^3 + X^4$:

```
sage : x = GF(2)['x'].gen()
sage : print lfsr_output(1+x^3+x^4,[1,0,1,1],n)
[1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0]
sage : print lfsr_sequence(coefficients,initial_state,n)==
lfsr_output(1+x^3+x^4,initial_state,n)
True
```

Chapitre 6

Manipulation de fonctions booléennes avec SAGE.

Une fonction booléenne est une fonction $f : GF(2)^n \rightarrow GF(2)$. Elle peut être représentée sous trois formes différentes :

1. table de vérité :
$$\{(x, f(x)), x \in GF(2)^n\}$$
2. forme algébrique normale : fonction polynomiale de n variables à coefficients dans $GF(2)$.
Cette écriture comme somme de monômes est unique :
$$f(x) = \sum_{u \in GF(2)^n} a_u \prod_1^n x_i^{u_i} \; avec \; a_u = \sum_{v \leq u} f(v)$$
3. forme polynomiale : trace d'un polynôme univarié à coefficients dans $GF(2^n)$.
Cette écriture peut être unique sous la forme trace suivante :
$$f(x) = \sum_{i \in \sigma_n} Tr^{\theta(i)}(a_i x^i) + \varepsilon(1 + x^{2^n - 1})$$
avec :
σ_n ensemble des représentants des classes cyclotomiques binaires mod $2^n - 1$
$\theta(i)$ le cardinal de la classe cyclotomique de i
$\varepsilon = wt(f) \, mod(2)$

Les fonctions booléennes se définissent dans SAGE à l'aide du constructeur `Boolean Function`.

```
sage : from sage.crypto.boolean_function import BooleanFunction
```

Les documentations et sources sont disponibles respectivement par la commande `BooleanFunction?` et `BooleanFunction??`

Il existe naturellement trois méthodes pour définir une fonction booléenne :

La première utilise une table de vérité, la liste des images des 2^n éléments de $GF(2)^n$, rangés dans l'ordre des entiers qu'ils représentent en little endian (bit de poids faible à gauche) :

```
sage : n=3
sage : truth_table = [0,1,0,0,0,0,0,0]
sage : f = BooleanFunction(truth_table)
sage : print [(i,f(i)) for i in range(2^n)]
[(0,False), (1,True), (2,False), (3,False),
(4,False), (5,False), (6,False), (7,False)]
```

On peut vérifier la représentation little endian (bit de poids faible à gauche) des entiers :

```
sage : def rpad(x,n) :
.... :      return Integer(x).digits(2,padto=n)
sage : print [(rpad(i,n),f(rpad(i,n))) for i in range(2^n)]
```

```
[([0,0,0],False), ([1,0,0],True), ([0,1,0],False),
([1,1,0],False), ([0,0,1],False), ([1,0,1],False),
([0,1,1],False), ([1,1,1],False)]
```

La table de vérité d'une fonction booléenne de n variables peut aussi être mise sous la forme d'une chaîne de 2^{n-2} caractères hexadécimaux, rangés dans l'ordre inverse de la table de vérité précédente :

```
sage : f_hex = BooleanFunction("a1")
sage : f_int = BooleanFunction([1,0,0,0,0,1,0,1])
sage : f_hex == f_int
True
```

Il existe donc plusieurs formats de représentation de la table de vérité :

```
sage : print f.truth_table(format='bin')
(False, True, False, False, False, False, False, False)
sage : print f.truth_table(format='int')
(0, 1, 0, 0, 0, 0, 0, 0)
sage : print f.truth_table(format='hex')
02
```

On note bien l'ordre inversé entre le format hex et le format int.

La deuxième façon de définir une fonction booléenne de n variables utilise sa forme algébrique normale, un polynôme booléen à n variables :

```
sage : R = BooleanPolynomialRing(3,'x')
sage : R.inject_variables()
sage : f = BooleanFunction(x0+x1+x2)
sage : f.algebraic_normal_form()
x0 + x1 + x2
```

La troisième façon de définir une fonction booléenne utilise sa forme polynomiale. La fonction booléenne de n variables est vue comme la trace d'un polynôme univarié à coefficients dans une extension de $GF(2)$ de degré n.

```
sage : K.<a> = GF(2^3,name='a')
sage : P.<x> = K['x']
sage : f = BooleanFunction(x^7+a*x+1)
```

Il n'existe pas dans Sage de méthode pour calculer la forme polynomiale d'une fonction booléenne. Une implémentation d'une forme polynomiale, sans unicité de la représentation, est proposée :

```
sage : def polynomial_form(f) :
.... :     n= f.nvariables()
.... :     K.<a>=GF(2^n,'a')
.... :     P.<x>=K['x']
.... :     return K['x'].lagrange_polynomial([(K(P(Integer(i).
.... :     digits(2))),GF(2)(f(i))) for i in range(2^n)])
sage : p = polynomial_form(f) ; print p
x^7 + (a^2 + a)*x^4 + a^2*x^2 + a*x + 1
sage : f == BooleanFunction(p)
True
```

On remarque que le polynôme trouvé n'est pas unique. En effet $a^3+a+1=0$ implique $a^4=a^2+a$. Comme 2 et 4 appartiennent à la même classe cyclotomique binaire modulo $2^n-1=7$, on voit que $Tr(a^4\times x^4)=Tr(a^2\times x^2)$. On en déduit

que $Tr(x^7 + ax + 1) = Tr(x^7 + (a^2 + a)x^4 + a^2x^2 + ax + 1)$.

```
sage : K.modulus()
x^3 + x + 1
```

On peut vérifier que la table de vérité de Tr(p) est bien la table de vérité de f :

```
sage : [ p(K(P(Integer(i).digits(2)))).trace() for i in range(2^n)]
== list(f.truth_table(format='int'))
True
```

L'entrée d'une fonction booléenne à n variables peut être un entier ou un binaire sur n bits en little endian (bit de poids faible à gauche) :

```
f(1) == f([1,0,0])
True
```

La sortie de la fonction booléenne est un booléen, qui peut être converti en élément de GF(2) :

```
print f(0),GF(2)(f(0))
True 1
print f(1),GF(2)(f(1))
False 0
print f(0)+f(1)
1
```

Il existe plusieurs opérations sur les fonctions booléennes : complément, addition, multiplication, concaténation.

La première opération est le complément de la table de vérité :

```
sage : print f.truth_table(format='int')
(1, 0, 0, 0, 1, 1, 1, 1)
sage : print (~f).truth_table(format='int')
(0, 1, 1, 1, 0, 0, 0, 0)
sage : print polynomial_form(f)
x^7 + (a^2 + a)*x^4 + a^2*x^2 + a*x + 1
sage : print polynomial_form(~f)
x^7 + (a^2 + a)*x^4 + a^2*x^2 + a*x
```

La deuxième opération est l'addition de fonctions booléennes, correspondant à l'addition des tables, des formes algébriques normales ou des formes polynomiales :

```
sage : (f+(~f)).truth_table(format='int')
sage : (1, 1, 1, 1, 1, 1, 1, 1)
sage : g = BooleanFunction(x^7)
sage : h = BooleanFunction(a*x+1)
sage : polynomial_form(g+h)
x^7 + (a^2 + a)*x^4 + a^2*x^2 + a*x + 1
```

La troisième opération est la multiplication de fonctions booléennes, correspondant à la multiplication des tables, des formes algébriques normales ou des formes polynomiales :

```
sage : (f*(~f)).truth_table(format='int')
(0, 0, 0, 0, 0, 0, 0, 0)
sage : polynomial_form(g*h)
```

```
x^7 + (a^2 + a)*x^4 + a^2*x^2 + a*x
```

On note que $x^7 \times (ax+1) = ax + x^7$sur $GF(2^3)$ et qu'on a a toujours $Tr((a^2+a) \times x^4) = Tr(a^2 \times x^2)$ sur $GF(2^3) \simeq GF(2)[x]/ < x^3 + x + 1 >$.

Enfin, on peut aussi concaténer des fonctions booléennes, correspondant à la concaténation des tables :

```
sage : print g.truth_table(format='int')
(0,1,1,1,1,1,1,1)
sage : print h.truth_table(format='int')
(1,1,1,1,0,0,0,0)
sage : print (g|h).truth_table(format='int')
(0,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0)
```

Attention, bien que ce ne soit pas précisé dans la documentation, il existe une limite pour le nombre de variables d'une fonction booléennes, liée à la mémoire de la machine :

```
sage : R = BooleanPolynomialRing(38,'s')
sage : s = R.gens()
sage : print BooleanFunction(s[0])
Boolean function with 38 variables

sage : R = BooleanPolynomialRing(39,'s')
sage : s = R.gens()
sage : print BooleanFunction(s[0])
MemoryError
```

Par ailleurs, les fonctions implémentées dans Sage sont implicitement prévues pour un ordre monomial lexicographique, l'ordre monomial par défaut du constructeur `Boolean PolynomialRing`. Elles ne donnent pas le résultat attendu pour un autre ordre monomial :

```
sage : R = BooleanPolynomialRing(4,'x',order='degrevlex')
sage : R.inject_variables()
Defining x0, x1, x2, x3
sage : f = BooleanFunction(x0*x2*x3+x1*x2+x0)
sage : print f.algebraic_normal_form()
x0*x1*x3 + x1*x2 + x3
```

Bibliographie

[1] Y.Renard. Expérimentations cryptographiques avec Sage. Thèse professionnelle de Mastère spécialisé Sécurité des Systèmes Informatiques et des Réseaux, Telecom Paristech. 2010.

[2] Sage Mathematical Software, version 4.4.4. `http://www.sagemath.org`

[3] P.Zimmermann et al. Calcul mathématique avec Sage. 2010. `http://sagebook.gforge.inria.fr`